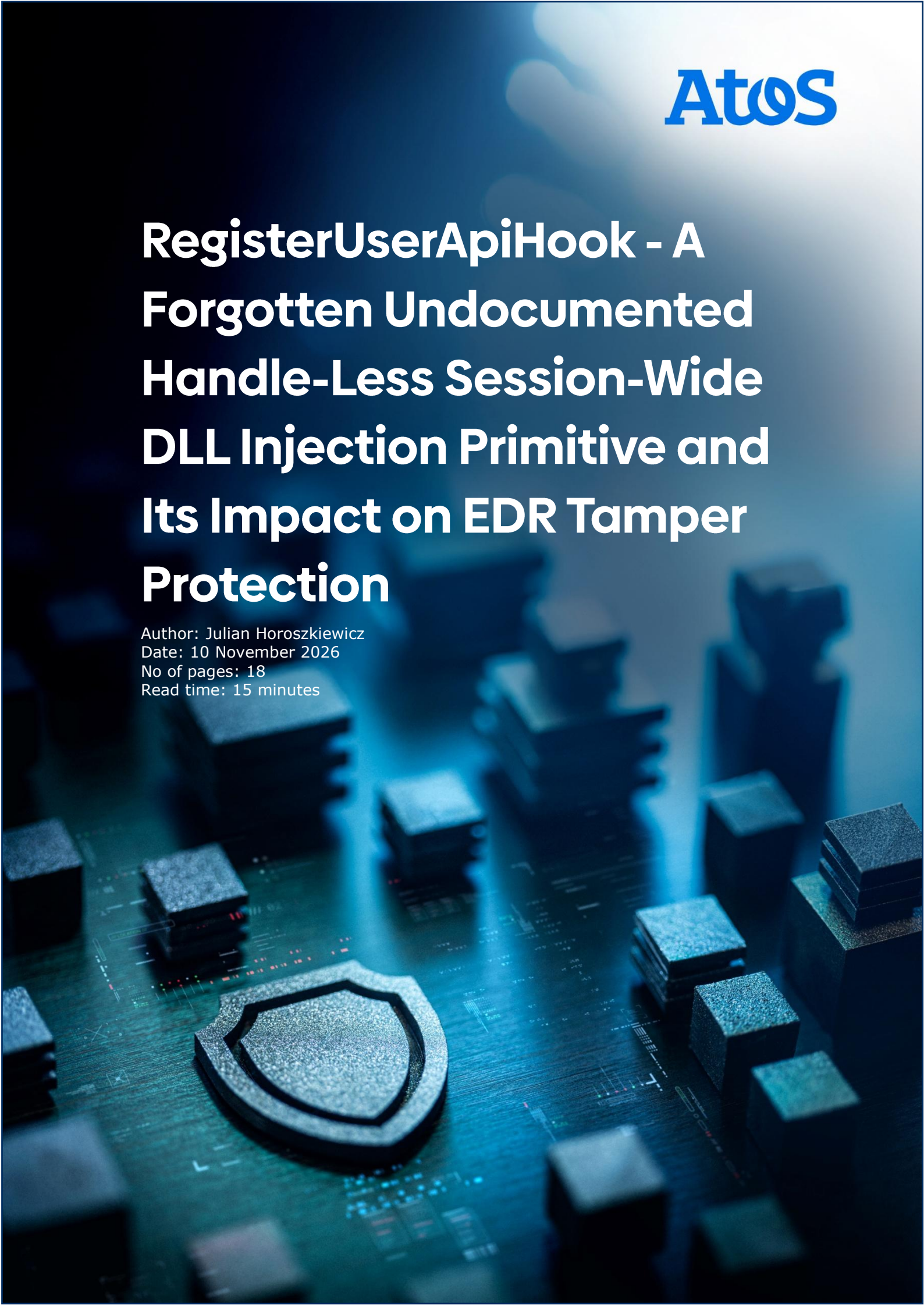


RegisterUserApiHook - A Forgotten Undocumented Handle-Less Session-Wide DLL Injection Primitive and Its Impact on EDR Tamper Protection

Author: Julian Horoszkiewicz
Date: 10 November 2026
No of pages: 18
Read time: 15 minutes

Table of contents

1 Introduction 3
1.1 Inspiration - SetWindowHookEx 3
1.2 Continuation - RegisterUserApiHook 4
1.3 The offensive potential 4
2 Demo 5
3 Technical details 5
3.1 API surface 5
3.2 Kernel Internals 7
3.3 Which processes are affected 10
3.4 Requirements Summary 12
4 The potential for offensive use: bypassing EDR tamper protection via trusted-context injection 12
4.1 The goal 13
4.2 What is being protected 13
4.3 Why RegisterUserApiHook is an efficient probe for the filter's exception list 15
4.4 Extending the sweep: invoking every executable in the install directory 16
4.5 Typical outcome: renaming the main service's executable without touching the PPL process 16
4.6 Results 16

1 Introduction

`RegisterUserApiHook` is an undocumented export from `user32.dll` that registers a DLL to be automatically loaded into every process in the same session that calls certain instrumented `user32.dll` exports (window/metrics/theming APIs). It is the mechanism Windows uses internally to inject `uxtheme.dll` (the theming engine) into GUI processes – ensuring that every window is themed without applications needing to opt in.

This API is a cousin of better-known SetWindowHookEx and SetWinEventHook. They all share the following important characteristics:

1. They provide DLL injection primitive into GUI-capable processes on Windows.
2. They do not require the caller to open a handle to target process.

The handle-less characteristic is significant in the context of EDR, especially when evaluating tamper protection. This is because **most of the kernel mode tamper protection logic is structured around object handles**.

1.1 Inspiration - SetWindowHookEx

A couple of years ago I discovered a local privilege escalation vulnerability in an undisclosed EDR product. The attack chain required to operate from a process that was trusted by the product's tamper protect filter.

There were two challenges:

1. While operating as a regular user with Medium integrity obviously I would not be able to open a writable handle to a process running with System integrity.
2. Standard process injection methods that involve handles obtained with OpenProcess were all blocked by the filter, which is standard behavior and was expected.

But there was one process that allowed me to work around both of those: the tray-embedded GUI component, a very common part of security software solutions.

Those usually run with Medium integrity and are usually also protected from standard process injection by the filter. Knowing that the filter callbacks were structured around handles, I started exploring other process injection primitives and decided to try with `SetWindowHookEx`. That turned out successful, allowing me to inject into that component, which itself was trusted by the filter, and as a result I was able to take advantage of the process trust relationship to carry out the rest of the attack.

1.2 Continuation - RegisterUserApiHook

This year I decided to revisit this subject, and search for more APIs that allow process injection without using a handle.

The outcome were 3 API calls: SetWindowHookEx, SetWinEventHook and RegisterUserApiHook.
Since RegisterUserApiHook is the least known among the three, I decided to explore it deeper.
The only mention of this method in the security context which I was able to find is this presentation from Recon 2012 - Smashing the Atom. As a reminder, the term "EDR" was officially coined in 2013.

1.3 The offensive potential

I have empirically confirmed this API to be a viable tampering approach alternative to BYOVD, by evaluating this idea against 4 well-known EDR products. For legal and compliance reasons I am not naming any vendors or products.

Contrary to SetWindowHookEx and SetWinEventHook, RegisterUserApiHook requires SYSTEM privileges (which is the same pre-requisite as for BYOVD).

To simply see what can be done with RegisterUserApiHook, feel free to skip the entire Technical details section and jump right into Potential for offensive use: bypassing EDR tamper protection via trusted-context injection.

2 Demo

A benign PoC can be found in the UxDiag/ directory and on the Github repository.

Tested on: Microsoft Windows 10.0.26200.9911 (win32kfull.sys version 10.0.26100.9711) and earlier.

3 Technical details

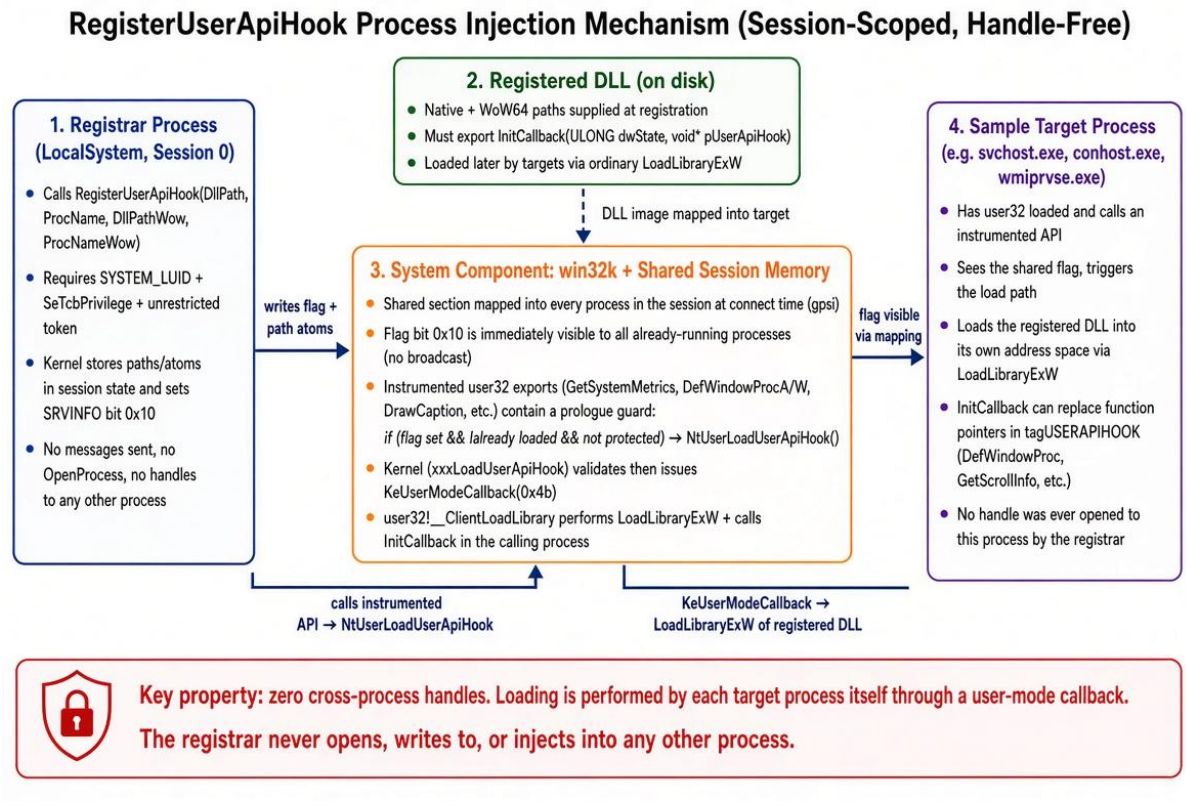


Figure 1 – RegisterUserApiHook mechanism

3.1 API surface

The API is session-scoped: a hook registered in Session 0 (the service session) affects only Session 0 processes, and a hook registered in the interactive session (Session 1, 2, etc.) affects only that session's processes. **Registration always targets the calling process's own session** – there is no parameter or flag to target another session (see "Registering for non-Session 0 sessions" below).

Only one hook can be registered per session at a time.

3.1.1 user32!RegisterUserApiHook

```
// Undocumented -- not in any public header
BOOL WINAPI RegisterUserApiHook(const USERAPIHOOKINFO* pInfo);
```

The input structure:

```
struct USERAPIHOOKINFO {          // Size = 0x28 (40 bytes)
    DWORD   Size;                 // +0x00  Must be 0x28
    DWORD   _pad;                 // +0x04  Alignment
    LPCWSTR DllPath;              // +0x08  Native (x64) DLL full path
    LPCWSTR ProcName;             // +0x10  Native export name (the init c
allback)
    LPCWSTR DllPathWow;           // +0x18  WoW64 (x86) DLL full path
    LPCWSTR ProcNameWow;          // +0x20  WoW64 export name
};
```

The wrapper in `user32!RegisterUserApiHook` converts the four strings into `UNICODE_STRING` structures via `RtlInitUnicodeString`, and forwards them to the kernel as (nativeDll, nativeProc, wowDll, wowProc) through `NtUserRegisterUserApiHook`.

Returns non-zero on success.

3.1.2 user32!UnregisterUserApiHook

```
BOOL WINAPI UnregisterUserApiHook(void);
```

Thin thunk to `NtUserUnregisterUserApiHook`. Can only be called by the same process that registered the hook.

3.1.3 win32u!NtUserLoadUserApiHook (no parameters)

```
// No user32 wrapper -- called directly from inside instrumented user32 ex
ports
LONG NtUserLoadUserApiHook(void);     // win32u syscall 0x14CC in this bui
ld
```

The no-parameter entry that causes the **current process** to load the session's registered hook DLL and run its init callback. It is invoked by user32.dll itself when it detects that a hook is registered for the session and the module has not yet been loaded in the calling process (see "Hook loading" below). Historical notes describe this being reachable via NtUserCallNoParam with an SFI_LOADUSERAPIHOOK index on older builds; in this Windows 11 build it is its own exported syscall.

3.1.4 DLL Init Callback

The DLL specified in the registration must export a function matching this signature:

```
BOOL WINAPI InitCallback(ULONG dwState, void* pUserApiHook);
```

The dwState argument selects one of three operations:

- 0 – Initialize.pUserApiHook points to a tagUSERAPIHOOK struct (0xE8 bytes) pre-filled with the real implementations by user32!ResetUserApiHook. The callback may replace function pointers (e.g., DefWindowProcA/W, GetScrollInfo, DrawCaption, etc.) to intercept window operations. Must return TRUE, must not zero the size field at offset 0x00, and must not alter the ForceResetUserApiHook pointer at offset 0xA8 (all three are validated by user32!InitUserApiHook).
- **1 – Uninitialize (normal).** Called with a NULL second argument – e.g. when a re-entrant load of a *different* module fails, or on the clear/reset paths.
- **2 – Uninitialize (forced).** Same as 1 but triggered by a force-reset path.

A minimal callback that simply returns TRUE for state 0 (without modifying the struct) is sufficient to get the DLL loaded. The struct's default function pointers remain intact, so normal window dispatch is unaffected.

3.2 Kernel Internals

3.2.1 Registration path

```
user32!RegisterUserApiHook
  -> win32u!NtUserRegisterUserApiHook       (syscall 0x1519)
    -> win32kfull!NtUserRegisterUserApiHook  (enters critical section)
      -> win32kfull!_RegisterUserApiHook     (validation + storage)
```

win32kfull!NtUserRegisterUserApiHook is a thin wrapper: EnterCrit -> _RegisterUserApiHook -> LeaveCrit. **It performs no broadcast and sends no message to existing processes** – verified by full decompilation of both functions in this build. Notification of already-running processes happens purely through the shared session flag (see "Flag propagation" below).

_RegisterUserApiHook (at win32kfull+0x27f81c) performs the following checks in order. Any failure produces ERROR_ACCESS_DENIED (5):

1. **IsInsideUserApiHook() must return 0.** session+0xff80 (native module index) must be -1, meaning no hook is currently registered in this session. Verified: IsInsideUserApiHook() is exactly ~*(uint*)(W32GetUserSessionState()+0xff80) >> 31.
2. **IsRestricted(currentThread) must return FALSE.** Calls win32kbase!IsRestricted, which calls SeTokenIsRestricted() on the process token. **Restricted tokens are rejected.** This is why PsExec -s fails – it can produce a restricted SYSTEM token.

3. **W32Process+0x300 must equal SYSTEM_LUID.** The process token's authentication LUID must be {0x3E7, 0} (both halves compared: +0x300 and +0x304 against luidSystem).
4. **HasTcbPrivilege() must return TRUE.** The process must hold SeTcbPrivilege (Act as part of the operating system).

3.2.2 Flag propagation (no broadcast)

The SRVINFO flags word is not re-sent to clients by any message. Instead:

1. Each win32k session owns a shared-memory section (session+0x5220).
2. When a process first connects (NtUserProcessConnect), win32kfull!InitMapSharedSection maps that section into the process with MmMapViewOfSection (view bases cached in W32Process+0x2E8/+0x2F0) and hands user mode pointers into the mapping via the 0x248-byte _USERCONNECT response block.
3. user32.dll stores that block at process initialization (_UserClientDllInitialize / ClientThreadSetup) and sets its global gpsi to point into the mapped section.
4. Because every process in the session maps the *same* section, when _RegisterUserApiHook writes bit 0x10 under lock, **all already-running processes in the session see it immediately** through their own mapping – no per-process update, no broadcast message.

Existing processes then pick up the flag lazily the next time they call an instrumented user32 export (see "Hook loading").

3.2.3 Unregistration path

_UnregisterUserApiHook (at win32kfull+0x296af0) checks that the calling process's W32Process pointer matches session+0xff78 (the process that registered the hook); otherwise ERROR_ACCESS_DENIED. Only the registering process can unregister. On success it clears session+0xff78, calls RemoveHmodDependency() on both module slots, resets the indices to -1, and clears bit 0x10 of the shared SRVINFO word (mask 0xFFFFFFEF).

Note the asymmetry: if the registering process exits without unregistering, the hook **stays registered for the rest of the session's lifetime** (the module refcounts keep the hmod table entries alive). There is no automatic cleanup on process exit.

3.2.4 Injectable exports (hook loading, per-process injection)

The trigger lives in **user mode**, not in the kernel. A fixed set of user32.dll exports contains a prologue guard that fires once per process. Verified direct callers of win32u!NtUserLoadUserApiHook inside user32 in the build I tested:

`DefWindowProcA`, `DefWindowProcW`, `GetSystemMetrics`, `GetSystemMetricsForDpi`, `GetScrollInfo`, `SetScrollInfo`, `EnableScrollBar`, `DrawCaption`, `DrawFrameControl`, `MDIRedrawFrame`, `SystemParametersInfoA`, `SystemParametersInfoW`,

SystemParametersInfoForDpi, _AdjustWindowRectEx, SetWindowRgn, and the internal __xxxClientBroadcastThemeChange.

Here is an example of the prologue guard, in GetSystemMetrics:

```
if (gfServerProcess == 0 && // not a server process (winlogon/csrss-style)
    !IsInsideUserApiHook() &&  // hook not yet loaded in THIS process
    (*gpsi & 0x10) != 0 && // SRVINFO_APIHOOK set for this session (shared mem)
    RtlIsThreadWithinLoaderCallout() == 0) {// not inside a DllMain/loader callout
    NtUserLoadUserApiHook();// win32u syscall 0x14CC, no parameters
}
```

IsInsideUserApiHook() here is the user-mode version: gfUserApiHook != 0 && ghmodUserApiHook != 0

Kernel side, win32kfull!NtUserLoadUserApiHook is a thin EnterCrit/LeaveCrit wrapper around xxxLoadUserApiHook, which enforces, in order:

1. **IsInsideUserApiHook()** – session+0xff80 != -1; otherwise returns FALSE and nothing happens. This is why the syscall can be used as a probe for "is a hook registered in my session".
2. **PsIsProtectedProcess(EPROCESS) == 0** – Protected / protected-light processes are skipped entirely.
3. **PsGetWin32KFilterSet() != 5** – Processes running under win32k filter level 5 are skipped.
4. **Thread guard word tagTHREADINFO+0x210** – Bits 0x1000000D (0, 2, 28) must all be clear; this word also carries other per-thread state, so it is a multi-purpose guard rather than a simple one-shot flag.
5. **WOW64 selection** – If the thread is WoW64, the slot at session+0x1018c (WOW module index) is used; otherwise session+0xff80.

It then calls xxxLoadHmodIndex(index), which:

1. Checks the **per-process** client-library cache. If this process already loaded this hmod, the cached handle is returned immediately – no reload, no re-init.
2. Otherwise resolves the DLL path from the session hmod atom table.
3. Selects the registered export name: native slot -> session+0xff84, WOW slot -> session+0x10190.
4. Calls win32kfull!ClientLoadLibrary(dllPath, procName), which builds a callback buffer holding both strings and invokes **KeUserModeCallback(0x4b, ...)** – i.e., the load happens through the KUSER_SHARED_DATA user-mode callback mechanism, not through a queued message.
5. On success it stores the returned HMODULE in the per-process cache. On specific failure statuses it marks the slot as failed (-1) so the process does not retry endlessly.

The user-mode half of step 4 is user32!__ClientLoadLibrary, which calls LoadLibraryExW:

```
flags = AppModelPolicy check ? 0 : LOAD_WITH_ALTERED_SEARCH_PATH;
hModule = LoadLibraryExW(dllPath, NULL, flags);
if (hModule && procName) {
    pfn = GetProcAddress(hModule, procName);   // ANSI-converted name
    if (pfn && InitUserApiHook(hModule, pfn))  // <-- runs the DLL's
callback with dwState=0
        NtCallbackReturn(&hModule, 0x18);      // HMODULE goes back to the
kernel
    else FreeLibrary(hModule);                 // init failed -> unload,
report failure
}
```

user32!InitUserApiHook then:

- zeroes a 0xE8-byte tagUSERAPIHOOK, pre-fills it with the real implementations via ResetUserApiHook,
- calls the DLL's callback with dwState=0,
- validates the result (non-zero return, size field at +0x00 non-zero, ForceResetUserApiHook pointer at +0xA8 untouched),
- on success copies the struct into user32 globals: sets gfUserApiHook=1, ghmodUserApiHook=hModule, gcLoadUserApiHook=1, installs all the PTR_Real* dispatch pointers (DefWindowProcA/W, Get/SetScrollInfo, EnableScrollBar, AdjustWindowRectEx, SetWindowRgn, QuerySendMessage, GetSystemMetrics(ForDpi), SystemParametersInfo(A/W/ForDpi), DrawFrameControl, DrawCaption, MDIRedrawFrame) and up to three 0x80-byte "LiteHookMsg" arrays,
- on failure calls the callback again with dwState=1, NULL.

Everything from step 4 onward runs in the target process's own context. **No cross-process handle is ever needed**, and nothing touches the target's module list except a normal LoadLibraryExW of the registered DLL.

3.3 Which processes are affected

Any process that (a) has user32.dll loaded and (b) calls one of the injectable exports listed above will load the hook on its first such call, provided it is not protected, not under filter level 5, and the thread guard word permits it. Processes that never load user32 (pure console apps that avoid all user32 APIs) are never hooked. GUI-subsystem and COM/RPC-using processes in practice hit GetSystemMetrics/DefWindowProc-family calls very early, so they get hooked almost immediately after registration.

In practice, the following commonly get hooked in Session 0 (the list is not exhaustive):

```
svchost.exe (GUI subsystem, creates windows for COM/DCOM)
```

```
conhost.exe (GUI subsystem, is the console window host)
wmiprvse.exe (GUI subsystem, WMI provider host using COM)
WUDFHost.exe (GUI subsystem, user-mode driver framework)
SearchProtocolHost.exe (Console subsystem, but creates windows via COM)
SearchIndexer.exe (GUI subsystem)
```

3.3.1 Known Consumer: uxtheme.dll

The only known consumer of this API is uxtheme.dll!CThemeServices::InstallThemeHooks (at uxtheme+0x2d1c8). It:

1. Calls IsSystemProcessContext() to verify it is running in a system process context.
2. Builds the system directory path to uxtheme.dll (%SystemRoot%\System32\uxtheme.dll).
3. Calls RegisterUserApiHook with export name ThemeInitApiHook for both native and WoW64 slots.

ThemeInitApiHook replaces function pointers in the tagUSERAPIHOOK struct with themed versions (ThemeDefWindowProcA/W, ThemeGetScrollInfoProc, ThemeDrawCaption, etc.), which is how Windows applies visual themes to all windows without per-application opt-in.

I also empirically verified that an interactive (Console) session has this hook registered at logon time.

This registration persists for the lifetime of the session. In the interactive session it is registered by a system process during logon; no long-lived SYSTEM process is visible in the Console session afterwards (consistent with a transient logon-time process). Because only the registering process can call UnregisterUserApiHook and there is no cleanup when that process exits, **the interactive session's hook slot stays occupied until logoff**.

3.3.2 Registering for non-Session 0 sessions

Registration of this hook for sessions other than 0 (regular/interactive user sessions) is technically supported, but not with regular user's own token. And in practice the interactive session's slot is already taken by Windows' uxtheme hook.

The kernel gates registration on the *token*, not on the session:

- The token AuthenticationId must be SYSTEM_LUID ({0x3E7, 0}) – a regular interactive user's logon token has its own LUID and fails check #3 with ERROR_ACCESS_DENIED.
- SeTcbPrivilege must be enabled and the token must not be restricted.

But the hook is stored in **the calling process's own session** (W32GetUserSessionState()), with no way to name a different session. So:

- A **regular interactive user process** (any elevation short of a SYSTEM token) is rejected by the LUID check. UAC admin tokens do not help – the AuthenticationId is still the user's.
- A **LocalSystem service in Session 0** registers for Session 0 only. This is the PoC path.
- A **SYSTEM-token process running inside an interactive session** registers for *that* interactive session, affecting all of its processes.

Practical obstacles for this scenario are specifically:

1. **The slot is pre-occupied** by the uxtheme theme hook registered at logon (verified above). Registration fails with ERROR_ACCESS_DENIED until it is removed.
2. **Only the original registering process can unregister**, and that process is gone (transient, logon-time). There is no supported way to evict it from user mode short of ending the session (logoff) – or racing Windows: register your hook in that session *before* the theme hook gets installed during logon.
3. A SYSTEM process placed in the interactive session is visible and attributable; combined with the fact that every hooked process loads the DLL via a plain LoadLibraryExW of an absolute path, detection surface is similar to Session 0 usage.

3.4 Requirements Summary

- **Process token LUID == SYSTEM_LUID (0x3E7)**
- **Token must not be restricted** – SeTokenIsRestricted() check.
- **SeTcbPrivilege enabled** – HasTcbPrivilege() check.
- **No existing hook in the caller's session** – interactive sessions are pre-occupied by uxtheme at logon and cannot be evicted after the fact.
- **DLL must be accessible to target processes** – loaded via LoadLibraryExW (with LOAD_WITH_ALTERED_SEARCH_PATH unless AppModel policy says otherwise) in each target's context.
- **Target process constraints (load side)** – protected/PPC processes and win32k filter set 5 are skipped (PsGetWin32KFilterSet() != 5); the per-thread guard word at tagTHREADINFO+0x210 (bits 0x1000000D) can suppress the load on a given thread.

A real Windows service running as LocalSystem satisfies all of these for Session 0.

4 The potential for offensive use: bypassing EDR tamper protection via trusted-context injection

`RegisterUserApiHook` **does not require the caller to posess a handle to the target process** (just like SetWindowHookEx and SetEventWinEventHook), making all

three very good candidates for bypassing self-protection mechanisms in security solutions.

The **handle-less aspect is the key**, because explicit requests for opening handles targeting EDR processes are blocked by the EDR's anti-tampering filter driver if they do not originate from a process EDR trusts, such as its own processes. Therefore, these hooks **bypass the handle-level protection** and if not explicitly known and blocked by the EDR driver as well, they become an effective injection and bypass primitive.

I confirmed this empirically, this time focusing solely on RegisterUserApiHook.

The target class were commercial EDR agents and their self-protection (tamper) filters.

4.1 The goal

The goal is **not to defeat PPL or the minifilter head-on**. Instead, we want to **inject into a non-PPL process that the product filter trusts**. The filter might still be protecting that trusted process from being injected into via common handle-based callbacks, which RegisterUserApiHook bypasses.

4.2 What is being protected

The diagrams below depict the high-level structure of most EDR solutions:

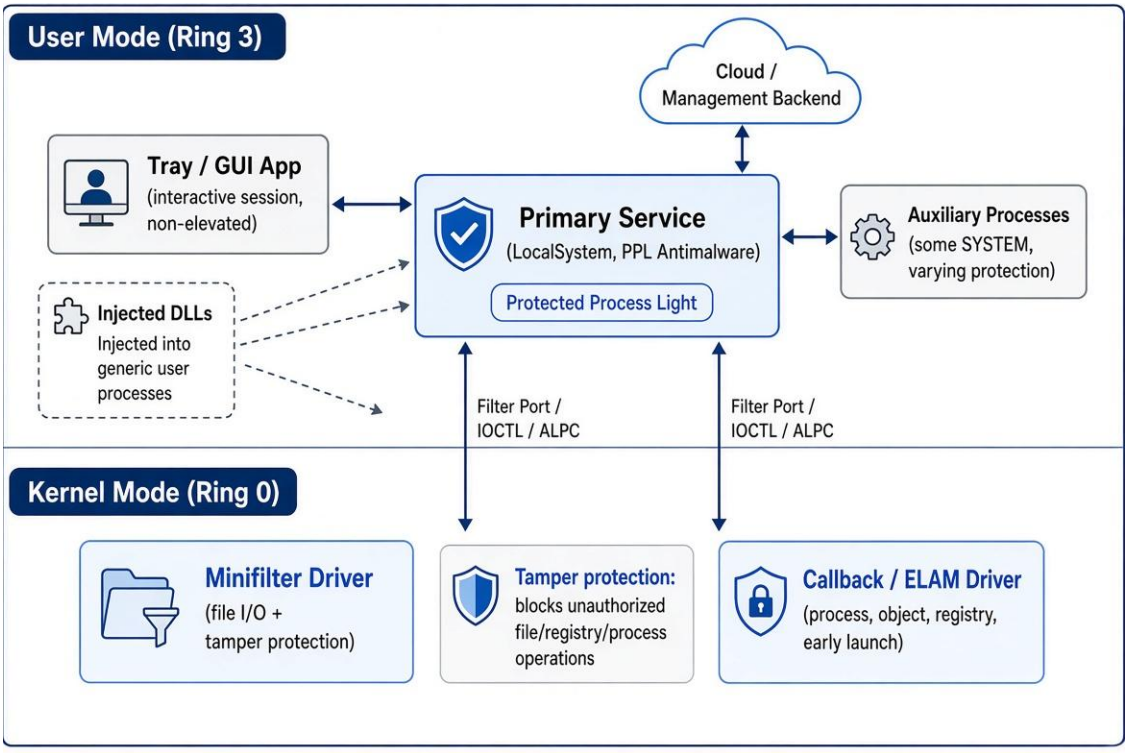


Figure 2 - SampleEDR logical architecture

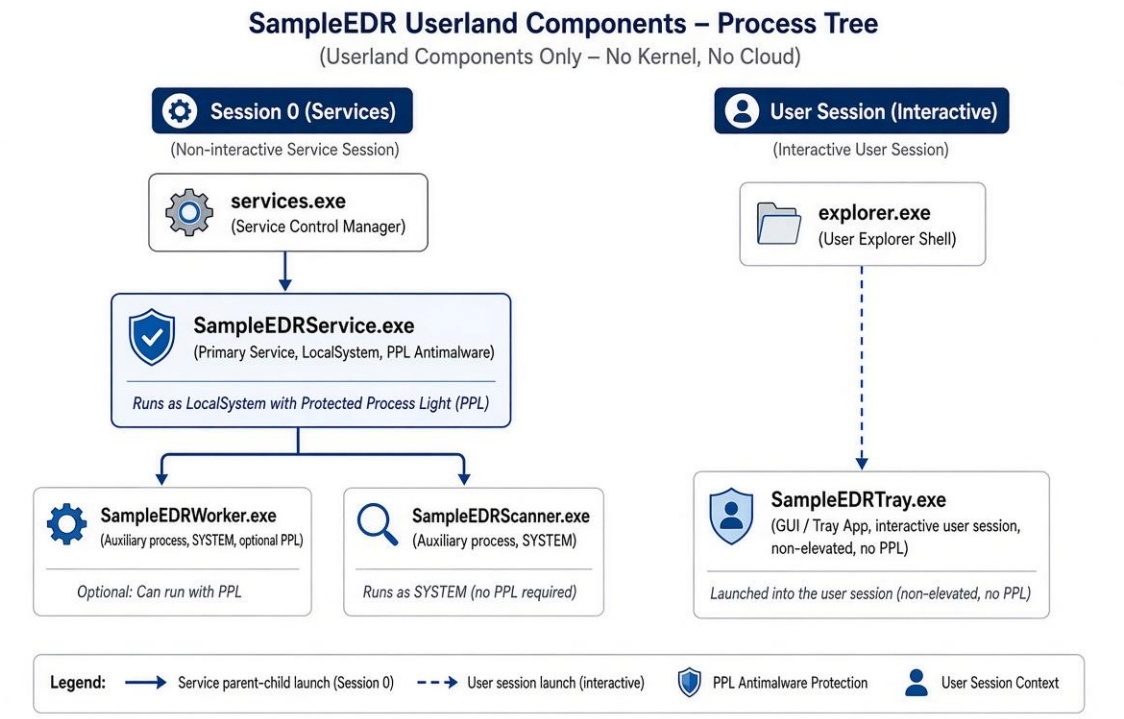


Figure 3 - SampleEDR process tree

A representative Windows EDR ("SampleEDR") layers its self-defense as follows:

Kernel mode: - A **file-system minifilter** in the I/O stack intercepts create/write/rename/delete on the product's installation files, configuration, and logs. This is the tamper-protection enforcement point, and its exception list typically matches on *caller* identity (image path / signature): the product's own processes may modify their own files, everyone else is blocked. - **Object callbacks** (ObRegisterCallbacks) strip dangerous rights (PROCESS_TERMINATE, PROCESS_VM_WRITE, …) from handles opened to protected processes; **process/thread creation and image-load notifications** feed detection; **registry callbacks** protect service configuration. - An **ELAM driver** enables the main service's launch as PPL and stores the certificate data that gates PPL module loading.

User mode: - The **primary service** (SampleEDRService.exe) runs as LocalSystem, in the common modern pattern launched as **Protected Process Light (Antimalware)**; it is the central orchestrator and a watchdog that restarts its auxiliaries. - **Auxiliary worker processes** (scanning, memory analysis, telemetry) – some SYSTEM/PPL, some lighter. - A **tray/UI component** in the interactive session: unprivileged (Medium integrity), usually also protected by the minifilter. - Optionally, **injected monitoring DLLs** in selected third-party processes for ntdll-level hooks.

The net effect: even a SYSTEM-level attacker is blocked from touching the install directory (minifilter), from obtaining VM-write handles on the main service (object callbacks + PPL), and from terminating it (PPL). Every protection is keyed on *who* performs the operation against *what*.

The implied weakness: trust is per-process. **If code we control executes inside a process that sits on the minifilter's exception list**, its file operations inherit that trust. If we are able to inject into such process, we can abuse that trust.

4.3 Why RegisterUserApiHook is an efficient probe for the filter's exception list

One of the ways to find which processes are trusted is reverse engineering. An alternative, opportunistic approach is to get our probe code running inside each candidate. RegisterUserApiHook (and its siblings) provides it with properties no classic injection technique has simultaneously:

1. **No handle is ever opened on any target.** The load happens in the target's own context (win32k -> KeUserModeCallback -> LoadLibraryExW inside the target). There is no OpenProcess/WriteVirtualMemory sequence for the EDR's object callbacks to strip or flag – and empirically, modern EDRs fail to detect this as process injection. Caveat: image-load *notifications* will still see a new module appear in each host; that telemetry shows a self-load attributed to the host itself, not a cross-process injection event.
2. **One registration sweeps the whole session.** From a LocalSystem service, a single RegisterUserApiHook call causes *every* user32-instrumented process in Session 0 to load our DLL on its next instrumented API call – including EDR auxiliary processes and other system services (svchost, wmiprvse, WUDFHost, SearchIndexer, … see "Which processes are affected"). We do not pick targets; the mechanism enumerates them for us.
3. **The payload can be a one-bit test.** When the hook is called, the PoC DLL method simply attempts to create a new file (path and name configurable), and log the current process details. Example:

```
[DIAG] diag is running. Current PID 3364 (C:\Windows\system32\wbem\wmiprvse
.exe)
Command line: C:\Windows\system32\wbem\wmiprvse.exe -secured -Embedding
[DIAG] StartLog(init) in PID 3364
[DIAG] diag is running. Current PID 29256 (C:\Windows\system32\conhost.exe)
Command line: \??\C:\Windows\system32\conhost.exe 0x4
[DIAG] StartLog(init) in PID 29256
[DIAG] diag is running. Current PID 9552 (C:\Windows\System32\vds.exe)
Command line: C:\Windows\System32\vds.exe
[DIAG] StartLog(init) in PID 9552
```

If we set the target path to the EDR product installation directory, the only results that get through represent the processes allowed by the filter.

4. **The outcome is an empirical map of the exception list.** Register once; every candidate process self-injects the probe.
5. **Duration** - keep in mind that the results are not immediate and thus not exhaustive. The hook only executes when the process actively calls an instrumented export. The longer the hook is active, the more instrumented calls it captures.

4.4 Extending the sweep: invoking every executable in the install directory

The hook only lands in processes that:

(a) load user32 and

(b) call an instrumented export,

so a passive sweep misses product executables that are pure console tools or never touch those APIs. This gap can be closed by actively launching them. We enumerate every .exe in the product install tree and run each one as SYSTEM, in Session 0, giving it time to initialize. Each launch is an experiment with two independent hypotheses:

- **Injectability** - does this binary load user32 and hit an instrumented API (GUI subsystem, COM/RPC usage, or even a stray GetSystemMetrics)? If so, the hook DLL lands in it automatically.
- **Trust** - if injected, does its file probe (or other potentially disruptive operation) succeed?

Oftentimes there is at least one injectable trusted process - a lightweight helper, CLI scanner, update agent, or a plain LocalSystem launcher service. They are rarely PPL-protected and usually sit on their own exception list.

4.5 Typical outcome: renaming the main service's executable without touching the PPL process

In practice the sweep oftentimes surfaces at least one non-PPL product component trusted by the minifilter - for example a plain LocalSystem parent/launcher service, or some other auxiliary component.

From inside that process (via our injected DLL), renaming the *main service's executable* succeeds: the file operation is attributed to the launcher's/helper's image, which is on the exception list, while the PPL protection of the engine process is never consulted - PPL guards the running process object (handles, VM access, termination), not the on-disk image.

The end state: the protected engine keeps running for now (its image is already mapped), but its watchdog can no longer relaunch it after a crash or restart, and the next boot starts nothing. The product is crippled by one of the processes it trusts, while that process can be injected into without ever opening a handle to it.

4.6 Results

By using this technique, I have confirmed **successful tamper-protect bypasses** in 2 out of 4 well-known EDR products tested. In both cases the injection reached at least one process the product trusted to make writes in an otherwise protected location.

Additionally, NONE of the tested EDR solutions flagged the use of RegisterUserApiHook as process injection or as any other form of injection, beyond the regular LoadLibrary event each time the PoC DLL was loaded into a process.

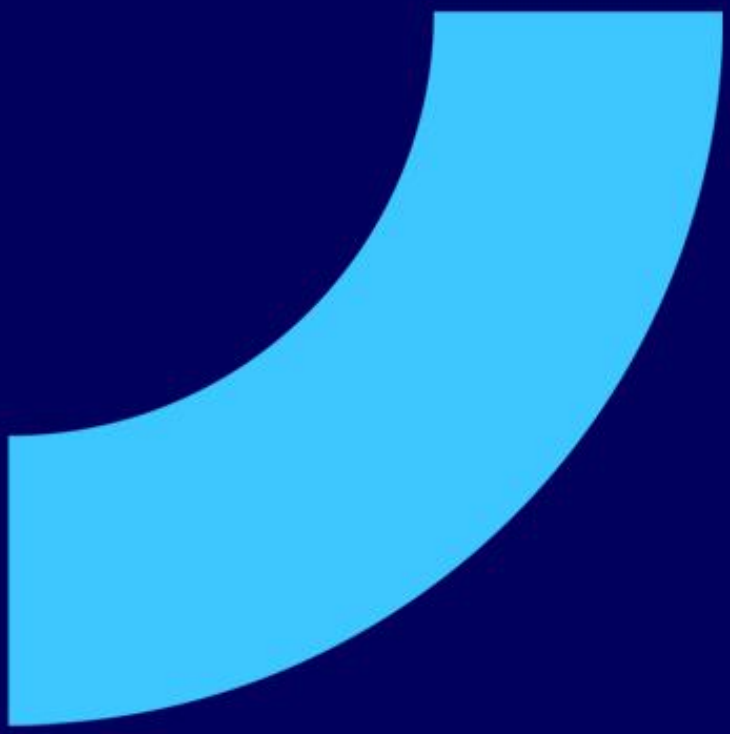
Therefore, I decided to share the results of this research exercise as potentially interesting for EDR vendors and defenders.

About Atos Group

Atos Group is a global leader in digital transformation with c. 56,000 employees and annual revenue of c. €7.2 billion (at the go-forward perimeter), operating in 54 countries under two brands - Atos for services and Eviden for products and systems. European number one in cybersecurity and a leader in cloud, Atos Group is committed to a secure and decarbonized future and provides tailored AI-powered, end-to-end solutions for all industries. Atos Group is listed on Euronext Paris.

Find out more about us

atos.net
atos.net/career



Atos Group is a registered trademark of Atos Group. © Atos Group. Confidential Information owned by Atos Group, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written

Atos